Continuous service execution in mobile prosumer environments

Unai Aguilera, Aitor Almeida,
Pablo Orduña, Diego López-de-Ipiña
DeustoTech
Avda. Universidades 24
48007 Bilbao, Spain
unai.aguilera,aitor.almeida,
pablo.orduna,dipina@deusto.es

Rafael de las Heras
Telefónica I+D
Emilio Vargas, 6
28043 Madrid, Spain
rafaelh@tid.es

Abstract

The vision of Ubiquitous Computing presents special requirements in mobile prosumer environments. Users have the possibility to create and provide their own services to others. Due to mobility, resources used to provide their services change and disappear continuously causing the disruption of the execution and the consumption process.

This paper proposes an architecture for mobile service prosuming which tackles the previous problems. It provides a decoupled vision of the different participants in the prosumer environment using various concepts: services, components and capabilities. These concepts enable a dynamic and continuous resolution process during prosuming and also the creation of composite services. This work presents an architecture for mobile devices which supports continuous service execution in a prosumer environment also including composite services. Components are resolved with available capabilities which are selected using restrictions performing a matching process. Due to the heterogeneity of computational resources in mobile devices, a dual approach for matching is proposed: semantic for powerful devices, and syntactic for limited ones.

1. Introduction

Nowadays, the introduction of mobile technologies in everyday life has increased dramatically and mobile devices have become an essential element for many people. Mobile devices enable not only to communicate with other users but also to access services which provide information and perform tasks for the user.

Traditionally, services have been provided by enterprises which offer their solutions to the users. However, in some cases these services are not adequate for user needs. For those situations the users should be offered with the possibility to create their own services which can resolve their current needs and problems. Moreover, those services could be provided in a prosumer way where the creators offer their services to other users [5], [9]. During provision from mobile devices, services access other resources in order to offer their functionality which can change (appearing, disappearing, or failing) due to the mobility of the user.

This paper proposes an architecture for user-empowerment which allows the execution of services in a mobile prosumer environment which also integrates composite services. This work is part of the mIO! project whose main goal is to research on technologies for mobility service provision. It introduces various concepts to represent the different elements which participate in the mobile prosumer environment and which are essential for the proposed architecture.

Heterogeneity in mobile computational resources is managed proposing a dual solution: devices powerful enough to apply semantic matching perform a component resolution process which uses semantic descriptions. However, limited devices use a syntactic approach.

The rest of the paper is organized as follows: Section 2 presents the basic concepts of the proposed service execution environment; Section 3 explains the architecture focusing on the continuous execution during mobility in Section 4. Service composition is explained in Section 5 while Section 6 presents the language used for service description. Finally, Section 7 presents the related work in the area and Section 8 summarizes the conclusions of the paper.

2. mIO! prosumer main concepts

The proposed prosumer environment relays on some concepts which are fundamental to the mIO! project vision: *mIO! services*, *components* and *capabilities*. Figure 1.shows a mIO! service which uses two components, a printer and a GSP, which are resolved using specific capabilities provided by hardware resources.

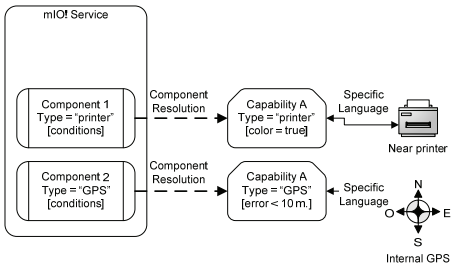

Figure 1. mIO! architecture concepts

2.1. mIO! services

Services which are created by users in the prosumer environment are called *mIO! services*. The creation process is performed by users using a tool which could be executed in the mobile device or in a computer. As result of the creation process a mIO! service template is obtained and published.

The *service template* contains information such as: general metadata about the service (author, creation date, etc.), logical and user interface rendering, and its functional description used during service composition. A template also contains a list of components required for mIO! service execution.

Service templates are instantiated during the execution process by the user's device. Templates can be executed in the same mobile device where created or published into repositories where they could be discovered and downloaded from in order to be executed by other users.

mIO! service templates not only contain information about the service part which is executed on the service provider mobile device but also the part which is downloaded and executed by the client side.

2.2. Components

Components are used to represent a functionality and are similar to *interfaces* in OO languages. They are used by the mIO! creation tool to help the users during the creation process selecting the service functionalities.

In addition, components enable to decouple the representation of the functionality that the user wants to add and the real implementation of that functionality. During the execution process the components are resolved using capabilities, which provide the actual functionality for the component. This process, which is called *Component resolution*, needs to be performed taking into account the mobility of the user and will be explained in Section 4. A component and a capability are compatible if they match at a functional level (methods signature). The mIO! architecture defines a taxonomy of components which can be used during service creation. The compatible capabilities will be also defined by the set of available components.

A component defines various elements which are used during the, creation, execution and resolution processes:

a) an interface with the mIO! service which includes the meta-data and logic needed by the creator to detect which components are compatible with the service being created. This information will also be used during execution in order to access the functionality provided by the component.
b) information about type which will be used during the resolution process to select compatible capabilities.
c) restrictions about the component which define non-functional properties and are included during creation time by the author. These restrictions enable to select a more adequate capability according to the service creator.

Components used by a mIO! service can be mandatory or optional for execution. Mandatory components must be resolved for correct service execution while optional services only add more functionality to the basic service.

During the execution of a mIO! service components can have two states: resolved (using a capability) or unresolved.

2.3. Capabilities

Capabilities are the implementations of those components used during the creation of mIO! services. Capabilities provide access to the functionality they represent and, in the meantime, hide the characteristics of the underlying resource. A capability can be implemented using a software or hardware resource and provides an abstract view of the actual resource (e.g. a map component resolved with Google Maps or Yahoo! Maps).

Capabilities include metadata information which defines their properties. This metadata is used during component resolution in order to select an adequate capability.

In the mIO! prosumer environment capabilities are classified using a location criteria: *Internal capabilities* are provided by the same device executing the mIO! service (e.g. GPS, camera, accelerometers, etc.). *External capabilities* are provided by other elements (external resources or other user's devices) and, therefore, remote communication is required in order to access them.

Capabilities have different communication interfaces: one for communicating with the execution architecture and the other with the resource. The first one provides the abstract and shared view of the resource while the other uses the specific language of the element being accessed. The conversion between these two languages is performed by the capability and the specific characteristics of the resource are hidden to the mIO! service.

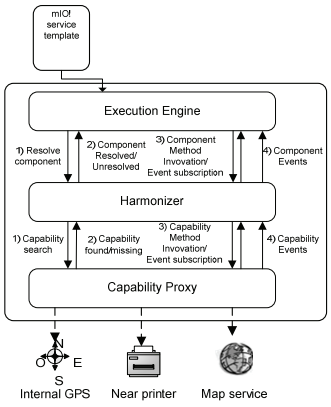

Figure 2. Execution architecture

3. mIO! service execution architecture

After a mIO! service has been created its template can be instantiated in a mobile device and start the execution of the service. A general view of the execution architecture is showed in Figure 2.
The main modules of the architecture are:

- *Execution Engine:* it executes the logical and visual representation of the service using the components specified in the mIO! service template. It also manages the provision of the client side logic and user interface. When a service is consumed by a mobile device, its local Execution Engine is responsible for executing the client logic and visual representation. Components contained in the template are used by the service logic but need to be resolved in actual capabilities before any invocation can be performed. The execution engine controls the life-cycle of the service: starting, suspending or stopping the execution in response to different events. For example, if a component, which is mandatory for the mIO! service execution, cannot be resolved, the Execution Engine suspends the service until the component could be resolved.
- *Harmonizer:* it performs the component resolution tasks in a dynamic (during service execution) and continuous way (during all service's life cycle). The Harmonizer uses the information contained in the component description to search and select an adequate capability. The selection uses the type information contained in the component description and the restrictions included by the mIO! service creator. Capabilities change over time due to mobility of the users: they appear, disappear and there could be communication problems, error in the resources providing the capability, etc. In order to provide users with a continuous execution experience, the Harmonizer searches and changes the capabilities used to resolve a component in a dynamic, seamless and autonomous way. It repairs transparently the unlinked components, whenever possible, with other available and compatible capabilities. In addition, capabilities could be autonomously changed when a better compatible one is found. The harmonizer internal work is presented in Section 4.

- *Capability Proxy:* it provides access to the available capabilities. It offers a common way to discover and access capabilities using different protocols. This layer encapsulates the particularities of the heterogeneous service and communication technologies (i.e. Bluetooth, UPnP, etc.) with the usage of a common interface. Available capabilities are discovered and notified to the Harmonizer which uses them to resolve managed components. Internal (i.e. those directly provide by the user device) and external (i.e. provided by other devices) capabilities are managed by the Capability Proxy. Thanks to this common interface for discovering and accessing the different capabilities it is possible to easily change them during the mIO! service life-cycle minimizing the interruption during execution.

3.1. Motivation scenario

Let's suppose that a user wants to execute a Photo Album mIO! service which relays on a Printing component. The Execution Engine gets the mIO! service and passes the component information to the Harmonizer which starts the resolution process. Compatible capabilities are searched with the type of the component (i.e. Printing) and filtered using the component's restrictions and the user's preferences (e.g. color printers which are near to the user).

When a capability which fulfills the requirements is discovered it is used to resolve the printing component and the Execution Engine is notified. Any invocation done by the Execution Engine on the Printing component will be delegated to the underlying capability and events will be propagated. If the user moves to a location where the previous printer is not available the Harmonizer will start searching for new compatible printers. If none is located the Execution Engine will be notified in order to perform the corresponding activities (e.g. suspend the current mIO! service).

4. Dynamic and continuous component resolution

The usage of components and capabilities provides better ways of managing the life-cycle of services. Dynamic component resolution enables to instantiate mIO! services depending on user's mobility and optimize their execution. In addition, it is possible to tackle common problems which arise during service execution such as communication problems or access issues by means of the capability substitution.

The component resolution is performed using capabilities which are available in the user's current context. These can be capabilities which are accessible by proximity (e.g. printers, screens, etc.) or capabilities which are globally accessible using telecommunication networks (e.g. 3G, GSM, etc).

The process is performed not only on the service provider side but also on the device which is consuming the service. It is possible that the client side of a mIO! service also needs to access capabilities meaning that the resolution process must be performed in both sides.

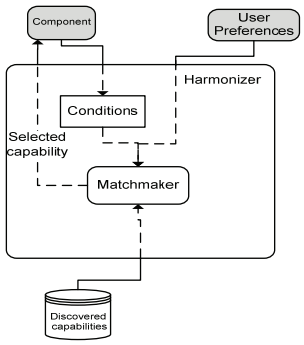

Figure 3. Harmonizer internal elements

The selection of capabilities is done taking into account the user's preferences, which are additional restrictions which could not be foresight by the service creator. For example, when selecting the printer needed by the Photo Album service the user is only interested in the cheapest and closest printers. The Harmonizer provides a way to specify component and user preferences that are used during the resolution process in order to select the best compatible capability. Those restrictions are expressed using a XML language explained in Section 4.2.

The component resolution is a continuous process which also comprises the monitoring of the resolved components. The process does not finish when a capability is resolved. Harmonizer

monitors changes in the user's context in order to detect when a capability must be changed for a better one. The whole process is performed in way that minimizes the interruption of the executed services as long as possible.

The internal process of the Harmonizer module, which performs the dynamic component resolution, is depicted in Figure 3.

4.1. Component resolution life-cycle

The execution of mIO! services depends on the current user's context. It is possible that the user wants to execute a service which cannot be currently executed due to the lack of compatible capabilities.

When a mIO! service cannot be executed in the user's current context, due to the lack of capabilities which satisfy the component requirements, the service execution can be stopped or suspended. In the first case, the user will need to start the service again in order to execute it. However, if the service is suspended waiting for compatible capabilities to appear, the Harmonizer will be continuously monitoring to detect the appearance of new compatible capabilities. The new capabilities could appear during mobility (i.e. approximation to a compatible capability) or the addition of new resources to the current user's context.

The Execution Engine manages the life-cycle of the mIO! services which the user wants to execute. When the Harmonizer is asked to resolve a component the Execution Engine must indicate if the former has to wait for the appearance of new capabilities. In this case, the Harmonizer will use the Capability Proxy to search for type compatible capabilities. The discovered capabilities will be filtered using the restrictions includes in the component description and the user's preferences. If a compatible capability appears the Execution Engine will be notified. When all the required components are resolved with capabilities the service execution will be resumed. After a component is resolved using a compatible capability the Harmonizer creates an internal link which enables to forward the component's method execution to the underlying capability.

The underlying resources that provide the functionality for the capabilities usually produce events during their operation. The Harmonizer supports event subscription when resolving a component with a compatible capability. However, due to the mobility of the user and possible capability changes, it is necessary to have a mechanism that assures the event re-subscription when a change occurs.

When a mIO! service execution needs to be stopped the Harmonizer must be notified in order to release the selected capabilities. This process is required in order to remove any event subscription that was performed with the underlying capability.

4.2. Capability selection

The Harmonizer uses the Capability Proxy to discover those compatible capabilities with the type of the component being resolved. All components contain information about the specific type they belong to which is based on predefined component taxonomy.

The component type states that any compatible capability implements a set of methods which are signature-compatible with the component's ones. A strong relationship among components and capabilities has the drawback that the set of different capabilities is predefined by type. However, it reduces the complexity of the component-capability matching process and the method and event forwarding that will be explained in Section 4.3.

Whenever a new capability is found by the Capability Proxy, the Harmonizer is notified and starts the filtering process using the component restrictions and user's preferences. This way, a new set of restrictions is obtained, which is applied by the matchmaker to the preferences contained in the discovered capabilities' meta-data. These restrictions are represented using a logical expression language that enables to specify desired conditions over the capability properties.

The proposed restriction language is independent of the technology used to represent the capability properties and, therefore, of the matching process used to select the capabilities. This means that capability selection can be done using semantic or syntactic matching depending on the device capabilities. Those devices with limited computational resources which cannot perform expensive computing procedures will execute the syntactic matching to select the compatible capabilities. However, those user's

devices which are fully capable will use the conditions as part of a semantic query using the potential of semantic reasoners.

An example of the proposed language, expressed in XML, is showed in Figure 4. These conditions can be included in the mIO! service template for each used component.

```
<component id="gps" type="mio:GPS" mandatory="yes">
  <conditions>
     <and>
        <condition property="global.situation"
           operator="==" valor="mio:Internal"/>
        <condition property="global.location.granularity.type"
           operator="==" value="mio:Circle"/>
        <or>
           <condition property="global.location.units"
             operator="==" value="meters"/>
           <condition property="global.location.units"
            operator="==" value="miles"/>
        </or>
        <condition property="global.location.granularity.radius"
           operator="&lt;=" value="5"/>
     </and>
     <selector property="global.location.granularity.radius"
            operator="MIN"/>
  </conditions>
</component>
```

Figure 4. Conditions expressed using XML

Each condition refers to a specific property and establishes a restriction about its value. The language defines some operators: the usual comparison operators (i.e. ==, <=, >=, !=) and some selectors (*MAX*/*MIN)* which enable to filter the available results.

Conditions expressed in XML could be converted to other query languages such as SQL or SPARQL depending on the matching technology used. The XML language enables to decouple the restriction representation from a specific technology (e.g. semantic) and can be used in computational limited devices.

In the example conversion to SPARQL, shown in Figure 5, conditions are converted to a *WHERE* clause containing a *FILTER* part which includes the conditions about properties. *MAX* and *MIN* reserved words are implemented using an OPTIONAL combination that selects the max or min value, respectively.

The results of this selection process are used by the Harmonizer to resolve the component. This process is repeated using any type compatible capability that is found by the Capability Proxy until the component is unregistered by the Execution Engine.

Capability properties are expressed in a XML description which is augmented with semantic annotations. XML tags are annotated with concepts contained in an ontology which models different aspects of the mIO! environment.

Those devices enabled with a semantic reasoner will use these annotations to create a full semantic representation of the capability description. The SPARQL could then be applied using the reasoned to perform the matching process.

```
PREFIX mio: <http://www.cenitmio.es/ontologies/mio.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?id
WHERE {
   ?capability rdf:type mio:capability .
   ?capability mio:id ?id .
   ?capability rdf:type mio:GPS .
   ?capability mio:situation ?situation .
   ?capability mio:location ?location .
   ?location mio:units ?units .
   ?location mio:granularity ?granularity .
   ?granularity rdf:type mio:Circle .
   ?granularity mio:radius ?radius
   OPTIONAL {
       ?capabilityB rdf:type mio:capability .
       ?capabilityB mio:id ?idB .
       ?capabilityB mio:location ?locationB .
       ?locationB mio:granularity ?granularityB .
       ?granularityB mio:radius ?radiusB
       FILTER ((?radius > ?radiusB))
   }
   FILTER (!bound(?capabilityB)
      && ((?situation = mio:)
      && ((?units = "meters") || (?units = "miles"))
      && (?radius <= 5.0)))
}
```

Figure 5. Conditions expressed as SPARQL WHERE and FILTER clauses

An example of the properties describing a capability is shown in Figure 6.

```
<?xml version="1.0" encoding="UTF-8"?>
<capability
    xmlns:mio="http://www.cenitmio.es/ontologies/mio.owl#"
    typeof="mio:GPS" about="mio:GPS1">
   <id typeof="http://www.w3.org/2001/XMLSchema#string"
       property="mio:id">GPS1</id>
   <type typeof="http://www.w3.org/2001/XMLSchema#string"
         property="mio:type">mio:GPS</type>
   <situation typeof="mio:situation" property="mio:situation"
              about="mio:Internal">mio:Internal</situation>
   <location typeof="mio:location" property="mio:location"
             about="mio:GPS1_location">
      <granularity typeof="mio:Circle" property="mio:granularity"
                   about="mio:GPS1_Circle">
         <type typeof="http://www.w3.org/2001/XMLSchema#string"
               property="mio:type">mio:Circle</type>
         <radius typeof="http://www.w3.org/2001/XMLSchema#float"
                 property="mio:radius">5</radius>
      </granularity>
      <units typeof="http://www.w3.org/2001/XMLSchema#string"
             property="mio:units">meters</units>
   </location>
</capability>
```

Figure 6. Properties describing capability

4.3. Method invocation and event handling

Any invocation performed by the mIO! service using a component is captured by the Harmonizer and forwarded to the underlying capability. The results of method invocations are returned to the component and then to the service.

If a component invocation fails because a capability is currently missing or due to a communication problem, the Harmonizer could block the call until a compatible capability is

found. When the new capability is discovered the invocation is forwarded and the results returned to the mIO! service.

The resources which provide the capabilities could raise events during operation meaning that a subscription process must be performed by the event consumer (i.e. mIO! services). However, event handling presents some problems during capability changing. If the capability linked to a component is changed, and there were current subscriptions, they will be lost and the new capability will not have knowledge of them.

In order to resolve the subscription handling, the Harmonizer internally tracks the subscriptions performed by the mIO! services using the components. Each component instance, which is related to a unique service, contains a list of the active subscriptions. This way, when the capability is changed by the Harmonizer, the event subscriptions are performed again using the new capability. In order to avoid the maintenance of old subscriptions by resources the event subscriptions are only valid during some time and must be renewed periodically.

5. Execution of composite services

When a user wants to perform a task that cannot be carried out by executing any of the available mIO! service templates composition could be used to resolve the problem. The goal of service composition is to create a complex service using the functionality provided by other services. In a prosumer environment service composition means that the user can create a composite service which when executed will use services provided by other users.

Composite services could be created using a work-flow tool located in the user's device or in a personal computer. For these reason, there are two different moments when a composite service can be created by users: a) using the creation tool prior execution and publishing the resulting template. The usage of more powerful creation tools (e.g. using an authoring tool in a PC) than the user's mobile device enables to create composite services with more complex process logics. b) creation of composite services using the mobile device during user mobility to resolve a current need in that moment. A tool included in the user's device could enable to create the composite service by connecting services in a simpler way adequate to the limited interaction possibilities of the mobile device.

In any of the approaches explained before the result of the composition process is a mIO service template which can be processed by the Execution Engine. However, in this case, this template includes information about other mIO! services that must be discovered and the control logic needed to execute the defined work-flow in order to achieve the complex service execution.

5.1. Integration with component architecture

The mIO! service composition is integrated with the component and capabilities architecture. Services which take part in the composition process are encapsulated as capabilities. With this approach composite services can be managed by the execution architecture which has been explained before.

In order to fully integrate the mIO! service composition with the architecture some steps have been followed:

- Services created by users export some methods which can be used to perform the invocation. These methods must be annotated including the type of input and output parameters and will be used during the work-flow creation to filter and select compatible services.
- Create a capability to encapsulate mIO services. As other capabilities, which provide access to resources, this one provides access to a software resource which is implemented as a mIO! service. This process is done automatically when the service is used in a composition.
- Usage of a generic component that represents the access to mIO! services and is resolved with the capability explained before. A reference to the template of the mIO! service that takes part in the composition can be stored in the component description included in the composite mIO! service template. During execution, this component could be resolved with any capability provided by a mIO! service which is an instance of the desired template or to a specific instance of a compatible mIO! service if it is specified in the component.

5.2. Composite service execution

After a composite service has been created it can be executed using the architecture presented in Section 3. In order to execute those templates, the dependencies with those services which comprise the composite service must be resolved.

The template of compatible services to be used during resolution, or the specific mIO! service, is contained in the information which conforms the composite mIO! service template.

Two types of composite service execution have been identified in the prosumer environment: local and distributed execution. In the first case a mIO! service which takes part in the composition is executed in the same device that is executing the composite service. On the other hand, distributed execution means that some of the services which take part in the composition are executed on other mobile devices.

When all services are executed locally the life-cycle of those services is controlled directly by the local Execution Engine. This means that the mIO! service template of those services can be downloaded from a template repository and executed. On the other hand, if the execution of a composite service requires access to services provided by other devices the local execution architecture cannot have control over them. Only those services which are already being executed by other users in their devices could be accessed.

In both approaches the execution of a composite service requires the resolution of all dependencies (i.e. other mIO! services and their capabilities) during execution as has been explained in Section 4.

6. SDL

The SDL (Service Description Language) is the language used for the formal specification of a mIO! Service. The SDL is annotated semantically to be processed by devices with adequate capabilities, but in those more limited it can also be processed syntactically. Each mIO! Service has a server and client module. The server side is executed in the device that offers that service (i.e. one mobile device that gives the location of its owner) while the client side is executed by the consumers of the service. The description of the service is divided in three parts: the general metadata, the server description and the client description. The general metadata contains information which is common for both parts of the service like the creation date, information about the service creator, a user-readable description, etc. It also contains a functional description of the service, stating its methods and their inputs, outputs, pre-conditions and post-conditions. This functional description is used for both the component resolution process and the service composition.

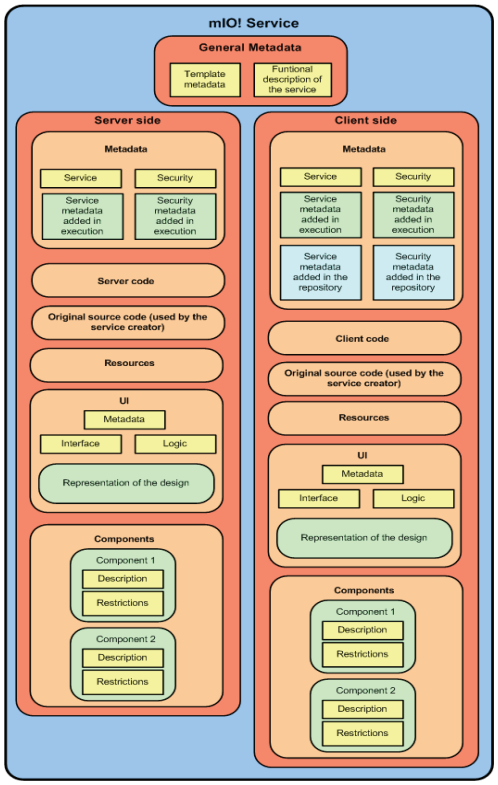

Figure 7. SDL conceptual representation

The server and client descriptions have a similar structure. They are composed of five parts:

- Metadata: Includes information about the service type, UUID, security certificates, and so on. This metadata can be added in the creation, when hosting the service in the repository or when the service template is instantiated.
- Code: The executable code.

- Source code: A copy of the source code to be used in the service creator.
- UI: Contains information about the user interface, including a renderizable specification.
- Components: Contains information about the components used in the service and the restrictions for their resolution.

Figure 7 shows a conceptual representation of the SDL.

7. Related work

This section presents previous work done in the area of service provision in mobile prosumer environments.

The future of service environments is addressed in [2], where the provision of micro-services in a prosumer environment is studied. This work assumes the existence of a number of telecommunication services, called service bricks which are used to the constructions of tradable micro-services or TMS. These services are traded among users with some monetary interchange. This approach applies self-management techniques to for controlling service trading and execution. The service brick concept is similar to the capability one presented in this paper, however, does not present an architecture for service prosuming.

The concept of service roaming used in [8] and [7] enables the consumption of services in a mobile environment Services are only valid in a specific scope becoming useless where the user moves outside it. Context changes are monitored to check for valid scopes during the execution triggering the search for other compatible services when current scope changes. Ontologies and service roaming are used in conjunction in [3]. However, none of these approaches takes into account the particular characteristics of micro-services in prosumer environments, where users create their own services which are provided to others.

The vision of autonomous systems which respond to changes and problems maintaining the provision of their services is called Autonomic Computing [10] and has many similarities with the dynamic component resolution presented in the present work. Its application to prosumer services has been addressed in [4].

None of the approaches cited before provide an integrated architecture for service composition and execution in a mobile prosumer environment. Service composition enables to create complex services and its application to Ubiquitous Computing is an active field of research [1]. An application of service composition to mobile environments is presented in [6]. This work addresses the problem of service disruption during mobility selecting those service providers which collectively deliver the composite service with the highest reliability. However, this approach does take into account the integration with a prosumer environment where users create, provide and consume micro-services.

There are several ready-to-use reasoning engines for mobile devices. Pocket KrHyper [11] is an open source reasoner for J2ME with support for propositional, first order and description logics. Witmate [12] in the other hand only supports propositional logic. Some authors [13][14] have tried to improve the reasoning algorithms for mobile devices.

8. Conclusions and future work

In the achievement of mobile prosumer environments there is a need for architectures which provide an integrated solution to service creation and execution. In addition, these architectures must take into account the possibility of executing compositions which enable to reuse services created by other users.

This paper presents an execution architecture, which is part of the mIO! project, that enables to execute services which are created by users in a prosumer environment. The proposed architecture offers an abstract view of the underlying resources and enables to provide a continuous execution despite the user mobility. In this architecture mIO! services are created using components. The architecture includes the Harmonizer layer whose task is to resolve components dynamically using compatible capabilities. Due to user's mobility the resolved components must be monitored in order to respond to possible events: capabilities appearing, disappearing and failures.

The capability selection process is performed in the mobile device, and due to the limited resources that these devices usually have, the application of semantic reasoning cannot be

always applied. This paper presents a restriction specification language which is independent of the used matching technologies. Restrictions for capability selection are expressed in XML and transformed to the particular query language used by the capability matching process. The architecture can be implemented in the devices with heterogeneous computational resources performing a syntactic or semantic process depending on the specific matching module installed.

In addition, the presented work also integrates the execution of composite services in a way that is fully integrated with the mobile execution architecture.

Future work will include testing the proposed architecture in a real environment, including the provision of more complex services in order to test the strength and weakness of the proposed solution.

9. Acknowledgments

This paper has been generated from the results of the deliverables E3.5.1 and E3.6.1 of the mIO! project (http://www.cenitmio.es) supported by the CENIT Spanish National Research Program (CENIT-2008 1019).

References

[1] J. Bronstead, K. Hansen, y M. Ingstrup, "A Survey of Service Composition Mechanisms in Ubiquitous Computing," Second Workshop on Requirements and Solutions for Pervasive Software Infrastructures, UbiComp 2007 Workshops Proceedings, 2007.

[2] P.H. Deussen, E. Höfig, y A. Manzalini, "An Ecological Perspective on Future Service Environments," Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, IEEE Computer Society, 2008, pp. 37-42.

[3] N. Weißenberg, R. Gartmann, y A. Voisard, "An Ontology-Based Approach to Personalized Situation-Aware Mobile Service Supply," GeoInformatica, vol. 10, Mar. 2006, págs. 55-90.

[4] John Strassner, "Autonomic Orchestration of Future Networks to Realize Prosumer Services," Future Networks, International Conference on, 2009, pp. 152-156.

[5] G. Hofmann y G. Thomas, "Digital Lifestyle 2020," IEEE MultiMedia, vol. 15, 2008, pp. 4-7.

[6] L.D. Prete y L. Capra, "MoSCA: seamless execution of mobile composite services," Proceedings of the 7th workshop on Reflective and adaptive middleware, Leuven, Belgium: ACM, 2008, pp. 5-10.

[7] A. Chin y K. Kontogiannis, "m-Roam: A Service Invocation and Roaming Framework for Pervasive Computing," Proceedings of the 18th International Conference on Advanced Information Networking and Applications - Volume 2, IEEE Computer Society, 2004, pp. 385.

[8] R. Gartmann, B. Holtkamp, y N. Weissenberg, "Service Roaming in Mobile Applications," Proceedings of the 2005 IEEE International Conference on Services Computing - Volume 01, IEEE Computer Society, 2005, pp. 121-128.

[9] J. Vazquez y D. Lopez-de-Ipina, "Social devices: autonomous artefacts that communicate on the Internet.". Internet of Things 2008. First International Conference, IOT 2008, Zurich, Switzerland, March 26-28, 2008, Proceedings. Springer Lecture Notes in Computer Science, Vol. 4952.

[10] J.O. Kephart y D.M. Chess, "The Vision of Autonomic Computing," Computer, vol. 36, 2003, pp. 41-50.

[11] Kleemann T. Towards Mobile Reasoning. International Workshop on Description Logics (DL2006); 2006 May 30 - June 1; Windermere, Lake District, UK; 2006.

[12] Witmate www. witmate.com (2010)

[13] Steller L, and Krishnaswamy S,. Pervasive Service Discovery: mTableaux Mobile Reasoning I-SEMANTICS 2008, September, Graz, Austria, 2008

[14] Steller L, Krishnaswamy S, Cuce S, Newmarch J, Loke S. A Weighted Approach for Optimised Reasoning for Pervasive Service Discovery Using Semantics and Context. 10th International Conference on Enterprise Information Systems (ICEIS); 2008 12 - 16 June; Barcelona, Spain; 2008.